Project Summary

MARLON TAMER - CREATING GHOST-ENGINE-3D

1. INITIALIZING DIRECT3D

1.1) Create the ID3D11Device and ID3D11DeviceContext interfaces using the D3D11CreateDevice function

Initializing Direct3D begins by creating the Direct3D 11 device (ID3D11Device) and context (ID3D11DeviceContext). These two interfaces are the chief Direct3D interfaces and can be thought of as our software controller of the physical graphics device hardware; that is, through these interfaces we can interact with the hardware and instruct it to do things

- The **ID3D11Device** interface is used to check feature support, and allocate resources.
- The **ID3D11DeviceContext** interface is used to set render states, bind resources to the graphics pipeline, and issue rendering commands.

```
HRESULT D3D11CreateDevice(
  IDXGIAdapter              *pAdapter,
  D3D_DRIVER_TYPE           DriverType,
  HMODULE                   Software,
  UINT                      Flags,
  const D3D_FEATURE_LEVEL *pFeatureLevels,
  UINT                      FeatureLevels,
  UINT                      SDKVersion,
  ID3D11Device              **ppDevice,
  D3D_FEATURE_LEVEL         *pFeatureLevel,
  ID3D11DeviceContext       **ppImmediateContext
);
```

In Project:

```
bool GraphicsEngine::init()
{
        D3D_DRIVER_TYPE driver_types[]=
        {
                D3D_DRIVER_TYPE_HARDWARE,
                D3D_DRIVER_TYPE_WARP,
                D3D_DRIVER_TYPE_REFERENCE
        };
        UINT num_driver_types = ARRAYSIZE(driver_types);

        D3D_FEATURE_LEVEL feature_levels[]=
        {
                D3D_FEATURE_LEVEL_11_0
        };
        UINT num_feature_levels = ARRAYSIZE(feature_levels);

        HRESULT res = 0;

        for (UINT driver_type_index = 0; driver_type_index < num_driver_types;)
        {
                res =D3D11CreateDevice(NULL, driver_types[driver_type_index], NULL, NULL,
                feature_levels, num_feature_levels, D3D11_SDK_VERSION, &m_d3d_device,
                &m_feature_level, &m_imm_context);
                if (SUCCEEDED(res))
                        break;
                        ++driver_type_index;
        }
        if (FAILED(res))
        {
                return false;
        }

m_imm_device_context=new DeviceContext(m_imm_context);
```

1.2) Describe the characteristics of the swap chain by filling out an instance of the DXGI_SWAP_CHAIN_DESC structure

The next step in the initialization process is to create the swap chain. This is done by first filling out an instance of the **DXGI_SWAP_CHAIN_DESC** structure, which describes the characteristics of the swap chain we are going to create.

```
typedef struct DXGI_SWAP_CHAIN_DESC {
  DXGI_MODE_DESC   BufferDesc;
  DXGI_SAMPLE_DESC SampleDesc;
  DXGI_USAGE       BufferUsage;
  UINT             BufferCount;
  HWND             OutputWindow;
  BOOL             Windowed;
  DXGI_SWAP_EFFECT SwapEffect;
  UINT             Flags;
} DXGI_SWAP_CHAIN_DESC;
```

In Project:

```
bool SwapChain::init(HWND hwnd, UINT width, UINT height)
{
        ID3D11Device*device= GraphicsEngine::get()->m_d3d_device;

        DXGI_SWAP_CHAIN_DESC desc;
        ZeroMemory(&desc, sizeof(desc));
        desc.BufferCount = 1;
        desc.BufferDesc.Width = width;
        desc.BufferDesc.Height = height;
        desc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
        desc.BufferDesc.RefreshRate.Numerator = 60;
        desc.BufferDesc.RefreshRate.Denominator = 1;
        desc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
        desc.OutputWindow = hwnd;
        desc.SampleDesc.Count = 1;
        desc.SampleDesc.Quality = 0;
        desc.Windowed = TRUE;
```

BufferCount: The number of back buffers to use in the swap chain; we usually only use one back buffer for double buffering, although you could use two for triple buffering.

OutputWindow: A handle to the window we are rendering into.

UINT Width: desired back buffer width
UINT Height: desired back buffer height
DXGI_RATIONAL RefreshRate: display mode refresh rate

1.3) Query the IDXGIFactory instance and create an IDXGISwapChain instance

A swap chain interface (IDXGISwapChain) is created through an **IDXGIFactory** instance with the IDXGIFactory::**CreateSwapChain** method:

```
HRESULT CreateSwapChain(
  IUnknown               *pDevice,
  DXGI_SWAP_CHAIN_DESC *pDesc,
  IDXGISwapChain        **ppSwapChain
);
```

In Project:

```
bool SwapChain::init(HWND hwnd, UINT width, UINT height)
{
        ID3D11Device*device= GraphicsEngine::get()->m_d3d_device;

        DXGI_SWAP_CHAIN_DESC desc;
        ZeroMemory(&desc, sizeof(desc));
        desc.BufferCount = 1;
        desc.BufferDesc.Width = width;
        desc.BufferDesc.Height = height;
        desc.BufferDesc.Format = DXGI_FORMAT_R8G8B8A8_UNORM;
        desc.BufferDesc.RefreshRate.Numerator = 60;
        desc.BufferDesc.RefreshRate.Denominator = 1;
        desc.BufferUsage = DXGI_USAGE_RENDER_TARGET_OUTPUT;
        desc.OutputWindow = hwnd;
        desc.SampleDesc.Count = 1;
        desc.SampleDesc.Quality = 0;
        desc.Windowed = TRUE;

        HRESULT hr=GraphicsEngine::get()->m_dxgi_factory->CreateSwapChain(device, &desc,
        &m_swap_chain);
```

The necessary fix is to use the **IDXGIFactory** instance that was used to create the device. To get this instance, we have to proceed through the following series of COM queries. DXGI (DirectX Graphics Infrastructure) is a separate API from Direct3D that handles graphics related things like the swap chain, enumerating graphics hardware, and switching between windowed and full-screen mode.

In Project:

```
m_d3d_device->QueryInterface(__uuidof(IDXGIDevice),(void**)&m_dxgi_device);
m_dxgi_device->GetParent(__uuidof(IDXGIAdapter), (void**)&m_dxgi_adapter);
m_dxgi_adapter->GetParent(__uuidof(IDXGIFactory), (void**)&m_dxgi_factory);
```

Create swap chain and release instances:

```
HRESULT hr=GraphicsEngine::get()->m_dxgi_factory->CreateSwapChain(device, &desc,
&m_swap_chain);
```

```
bool GraphicsEngine::release()
{
        m_dxgi_device->Release();
        m_dxgi_adapter->Release();
        m_dxgi_factory->Release();
        m_imm_device_context->release();
        m_d3d_device->Release();
        return true;
}
```

1.4) Create a render target view to the swap chain's back buffer

We do not bind a resource to a pipeline stage directly; instead, we must create a resource view to the resource and bind the view to the pipeline stage. In particular, in order to bind the back buffer to the output merger stage of the pipeline (so Direct3D can render onto it), we need to create a render target view to the back buffer.

```
bool SwapChain::init(HWND hwnd, UINT width, UINT height)
{


ID3D11Texture2D* buffer = NULL;
        hr=m_swap_chain->GetBuffer(0, __uuidof(ID3D11Texture2D), (void**)&buffer);
        if (FAILED(hr))
        {
                return false;
        }

        hr=device->CreateRenderTargetView(buffer, NULL, &m_rtv);
        buffer->Release();

        if (FAILED(hr))
        {
                return false;
        }

        return true;
}
```

1. A pointer to the swap chain's back buffer is obtained using the **IDXGISwapChain::GetBuffer** method. The first parameter of this method is an index identifying the **particular back buffer we want to get** (in case there is more than one). In our demos, we only use one back buffer, and it has **index zero**. The second parameter is the **interface type of the buffer**, which is usually always a **2D texture (ID3D11Texture2D)**. The third parameter **returns a pointer to the back buffer**.

2. To create the render target view, we use the **ID3D11Device::CreateRenderTargetView** method. The first parameter **specifies the resource that will be used as the render target**, which, in the previous example, is the **back buffer** (i.e., we are creating a render target view to the back buffer). The second parameter is a pointer to a D3D11_RENDER_TARGET_VIEW_DESC. Among other things, this structure describes the data type (format) of the elements in the resource. If the resource was created with a typed format (i.e., not typeless), then this parameter can be null, which indicates to create a view to the first mipmap level of this resource (the back buffer only has one mipmap level) with the format the resource was created with. Because we specified the type of our back buffer, we specify null for this parameter. The third parameter returns **a pointer to the create render target view object**.

3. The call to IDXGISwapChain::GetBuffer increases the COM reference count to the back buffer, which is why we release it **(ReleaseCOM)** at the end of the code fragment when we are done with it.

1.5) Create the depth/stencil buffer and its associated depth/stencil view

The depth buffer is just a 2D texture that stores the depth information (and stencil information if using stenciling). To create a texture, we need to fill out a **D3D11_TEXTURE2D_DESC** structure describing the texture to create, and then call the **ID3D11Device::CreateTexture2D** method.

```
typedef struct D3D11_TEXTURE2D_DESC {
  UINT              Width;
  UINT              Height;
  UINT              MipLevels;
  UINT              ArraySize;
  DXGI_FORMAT       Format;
  DXGI_SAMPLE_DESC SampleDesc;
  D3D11_USAGE       Usage;
  UINT              BindFlags;
  UINT              CPUAccessFlags;
  UINT              MiscFlags;
} D3D11_TEXTURE2D_DESC;
```

In Project:

```
SwapChain::SwapChain(HWND hwnd, UINT width, UINT height)
{


        D3D11_TEXTURE2D_DESC depth_desc = {};
        depth_desc.Width = width;
        depth_desc.Height = height;
        depth_desc.Format = DXGI_FORMAT_D24_UNORM_S8_UINT;
        depth_desc.Usage = D3D11_USAGE_DEFAULT;
        depth_desc.BindFlags = D3D11_BIND_DEPTH_STENCIL;
        depth_desc.MipLevels = 1;
        depth_desc.SampleDesc.Count = 1;
        depth_desc.SampleDesc.Quality = 0;
        depth_desc.MiscFlags = 0;
        depth_desc.ArraySize = 1;
        depth_desc.CPUAccessFlags = 0;

hr = device->CreateTexture2D(&depth_desc, nullptr, &buffer);

        if (FAILED(hr))
        {
                return false;
        }

        hr = device->CreateDepthStencilView(buffer, NULL, &m_dsv);
        buffer->Release();

        if (FAILED(hr))
        {
                return false;
        }
}
```

MipLevels: The number of mipmap levels. Mipmaps are covered in the chapter on texturing. For creating the depth/stencil buffer, our texture only needs one mipmap level.

1.6) Bind the render target view and depth/stencil view to the output merger stage of the rendering pipeline

Now that we have created views to the back buffer and depth buffer, we can bind these views to the output merger stage of the pipeline to make the resources the render target and **depth/stencil buffer** of the pipeline:

In Project:

```
void DeviceContext::clearRenderTargetColor(SwapChain* swap_chain, float red, float green, float
blue, float alpha)
{
	FLOAT clear_color[] = {red,green,blue,alpha};

	m_device_context->ClearRenderTargetView(swap_chain->m_rtv, clear_color);

	m_device_context->ClearDepthStencilView(swap_chain->m_dsv, D3D11_ClEAR_DEPTH)|
	D3D11_CLEAR_STENCIL, 1.0f, 0);

	m_device_context->OMSetRenderTargets(1, &swap_chain->m_rtv, swap_chain->m_dsv);
}
```

The first parameter is the number of render targets we are binding; we bind **only one** here, but more can be bound to render simultaneously to several render targets (an advanced technique). The second parameter is a pointer to the first element in an array of **render target view** pointers to bind to the pipeline. The third parameter is a pointer to the **depth/stencil view** to bind to the pipeline.

1.7) Set the viewport

Usually we like to draw the 3D scene to the entire back buffer. However, sometimes we only want to draw the 3D scene into a subrectangle of the back buffer. The subrectangle of the back buffer we draw into is called the **viewport**.

In Project:

```
void DeviceContext::setViewportSize(UINT width, UINT height)
{
	D3D11_VIEWPORT vp = {};
	vp.Width = (FLOAT)width;
	vp.Height = (FLOAT)height;
	vp.MinDepth = 0.0f;
	vp.MaxDepth = 1.0f;
	m_device_context->RSSetViewports(1, &vp);
}
```

The MinDepth member specifies the minimum depth buffer value and MaxDepth specifies the maximum depth buffer value. Direct3D uses a depth buffer range of 0 to 1, so MinDepth and MaxDepth should be set to those values, respectively, unless a special effect is desired.
Once we have filled out the **D3D11_VIEWPORT** structure, we set the viewport with Direct3D with the **ID3D11DeviceContext::RSSetViewports** method. Sets a viewport that draws onto the entire back buffer:

1.8) Resize Window

You can use the **IDXGISwapChain::ResizeBuffers** method to handle window resizing. Before you call ResizeBuffers, you must **release** all outstanding references to the swap chain's buffers. Like the RenderTargetView m_rtv.

In Project:

```
void SwapChain::Swap_Resize(UINT width, UINT height)
{
	m_rtv->Release();
	m_dsv->Release();

	m_swap_chain->ResizeBuffers(0, width, height, DXGI_FORMAT_R8G8B8A8_UNORM, 0);

	LoadViews(width, height);
}
```

Loadview is like SwapChain::init( ) without createSwapChain. We initialize the Views with new size.

With the Message_Handler we can register a windows resizing.

In Project:

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg, WPARAM wparam, LPARAM lparam)
{
	switch (msg)
	{

	case WM_SIZE:
	{
		Window* window = (Window*)GetWindowLongPtr(hwnd, GWLP_USERDATA);

		if(window) window->onSize();
		break;
	}
```

2. Rendering

2.1) Vertices

3. Drawing

3.1) Vertex Buffer

4. Components

4.1) Input System

Now we want to read whether a key is pressed or released. We can do that wie MessageHandler.

In Project:

```
LRESULT CALLBACK WndProc(HWND hwnd,UINT msg, WPARAM wparam, LPARAM lparam)
{
	switch (msg)
	{

	case WM_SIZE:
	{
		Window* window = (Window*)GetWindowLongPtr(hwnd, GWLP_USERDATA);

		if(window) window->onSize();
		break;
	}

	case WM_KEYDOWN:
	{
		Window* window = (Window*)GetWindowLongPtr(hwnd, GWLP_USERDATA);

		if (window) window->onKeyDown((unsigned int)wparam);
		break;
	}

	case WM_KEYUP:
	{
		Window* window = (Window*)GetWindowLongPtr(hwnd, GWLP_USERDATA);

		if (window) window->onKeyUp((unsigned int)wparam);
		break;
	}
```

We can initialize all keys with a 0. When a key is pressed then the value will change to „1“.

In Project:

```
bool Input::init()
{
	for (int i = 0; i < 256; i++)
	{
		m_keys[i] = 0;
	}
	return true;
}

void Input::KeyDown(unsigned int value)
{
	// when key is pressed -> position = 1
	m_keys[value] = 1;
}

void Input::KeyUp(unsigned int value)
{
	// when release -> position = 0
	m_keys[value] = 0;
}
```

5. Source

* Frank D. Luna: „Introduction to 3D Game Programming with DirectX 11“
https://files.xray-engine.org/boox/3d_game_programming_with_DirectX11.pdf